EE-334
Digital System Design

Custom Digital Circuits

From Algorithms to Architectures
Tips & Tricks

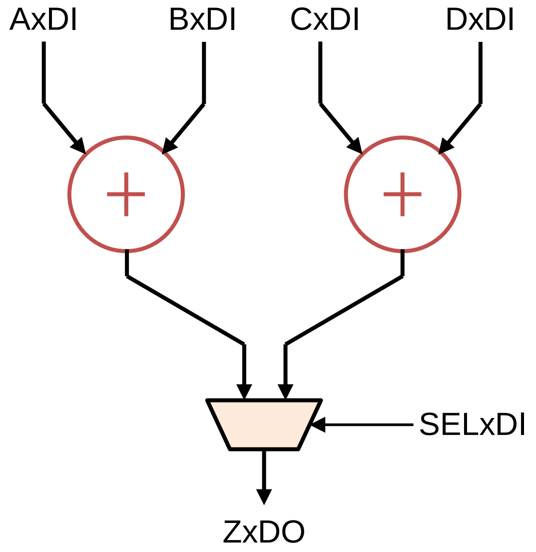
Andreas Burg

Resource Sharing for Area in Combinational Logic

- **Strict isomorphic mapping of an algorithm often implies redundant logic**
- **Example:** selection of one of multiple identical operations with different operands
  - Software: sequential execution naturally avoids unnecessary operations
  - Hardware: careless mapping generates redundant resources

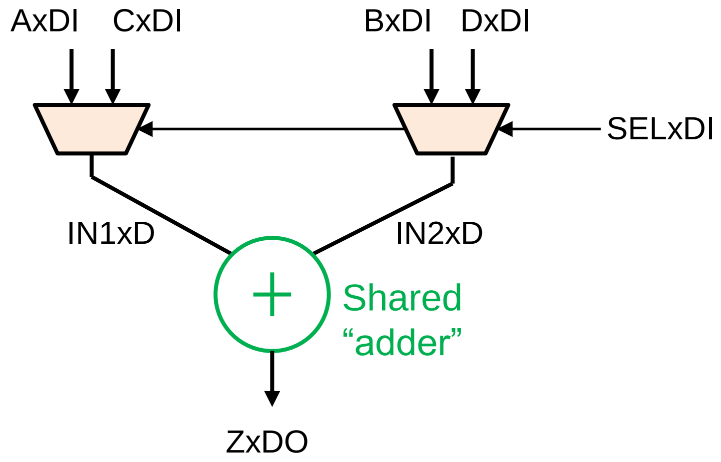
```
ZxDO <= AxDI + BxDI when SELxSI = ‘0' else
        CxDI + DxDI;
```

```
IN1xD <= AxDI when SELxSI = ‘0' else
         CxDI;
IN2xD <= BxDI when SELxSI = ‘0' else
         DxDI;
ZxDO <=  IN1xD + IN2xD;
```



Resource sharing by re-ordering selection and operations

In obvious cases performed automatically during RTL synthesis



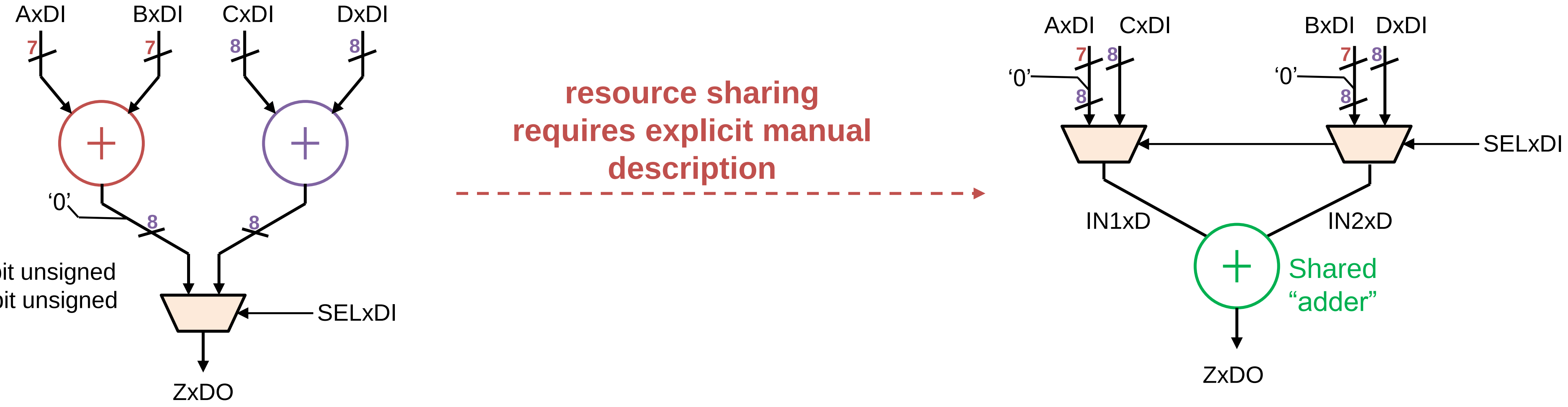
Resource Sharing for Area in Combinational Logic

- **RTL** synthesis performs **resource sharing only in obvious cases** (with fully identical operators)
- **Similar operators can often be shared manually** with little or no additional effort
- **Example:** different word lengths

```
ZxDO <= ‘0’ & (AxDI + BxDI) when SELxSI = ‘0' else
        CxDI + DxDI;
```

```
IN1xD <= ‘0’ & AxDI when SELxSI = ‘0' else
         CxDI;
IN2xD <= ‘0’ & BxDI when SELxSI = ‘0' else
         DxDI;
ZxDO <=  IN1xD + IN2xD;
```

AxDI, BxDI: 7 bit unsigned
CxDI, DxDI: 8 bit unsigned



Algebraic Transformations

- The mathematical **specification of an algorithm is** typically **not** an **unambiguous description of a suitable** isomorphic **datapath**
  - **Expressions can be re-written as other equivalent expressions**

- **Equivalent** (ideally more simple) **expressions can be derived using algebraic properties of the operators**

Commutativity	Associativity	Distributivity
implies symmetry with functional equivalence	**allows re-ordering of the same operator**	**allows re-ordering of different operators**
$a + b \leftrightarrow b + a$	$(a + b) + c \leftrightarrow b + (b + c)$	$d(a + b) \leftrightarrow da + db$

  - **Note:** **parentheses "(...)"** **and** **operator priorities** **define data dependencies** in the dataflow graph (i.e., in the isomorphic datapath)

Tree Re-Structuring for Timing with Associativity

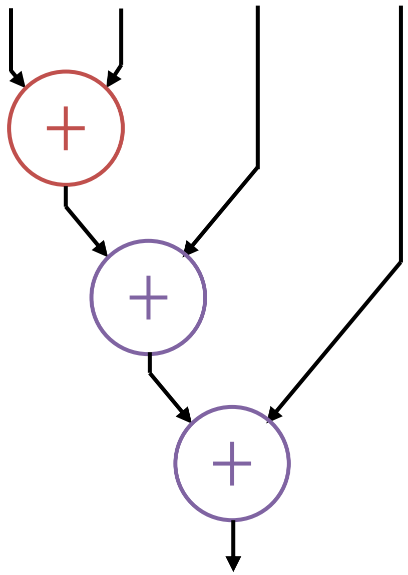
- **Objective:** reduce the longest path by avoiding data dependencies between intermediate results
- **Re-structuring:** prioritize operations to avoid/reduce data dependencies
  - **Changing the order using associativity**

$a + b + c + d$

Order not explicitly defined (only implicitly from left-to-right)

Linear (serial) structure

$\big((a + b) + b\big) + c$

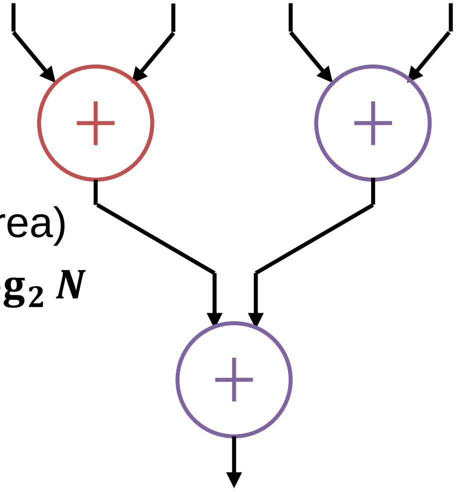


TREE Restructuring

- Same number of operations (area)
- Depth (**delay**) only grows as $\mathbf{\log_2 N}$
- Balanced delays

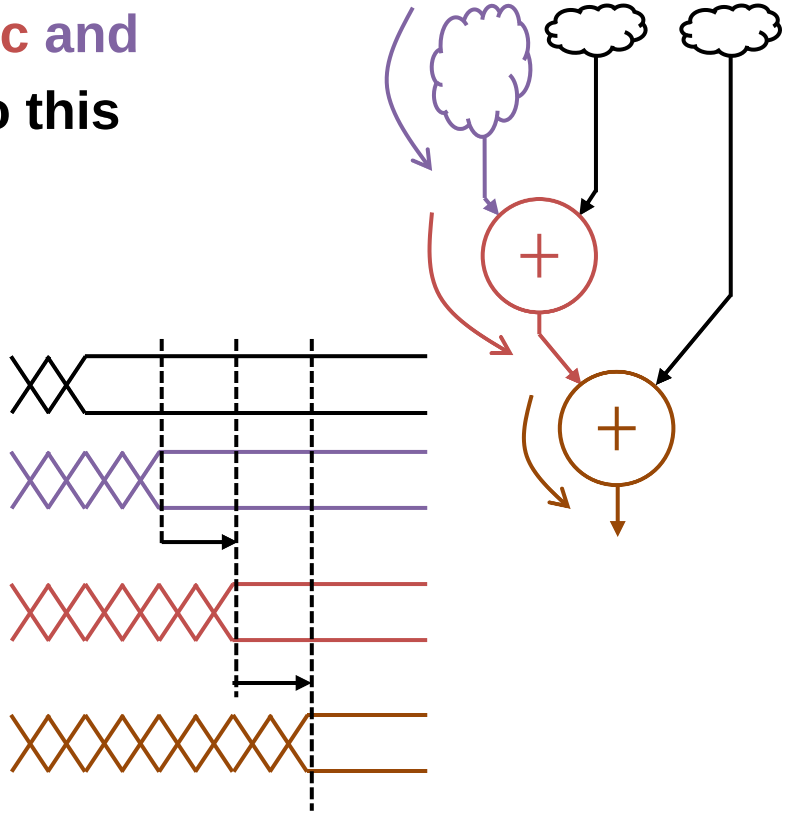
Tree (parallel) structure

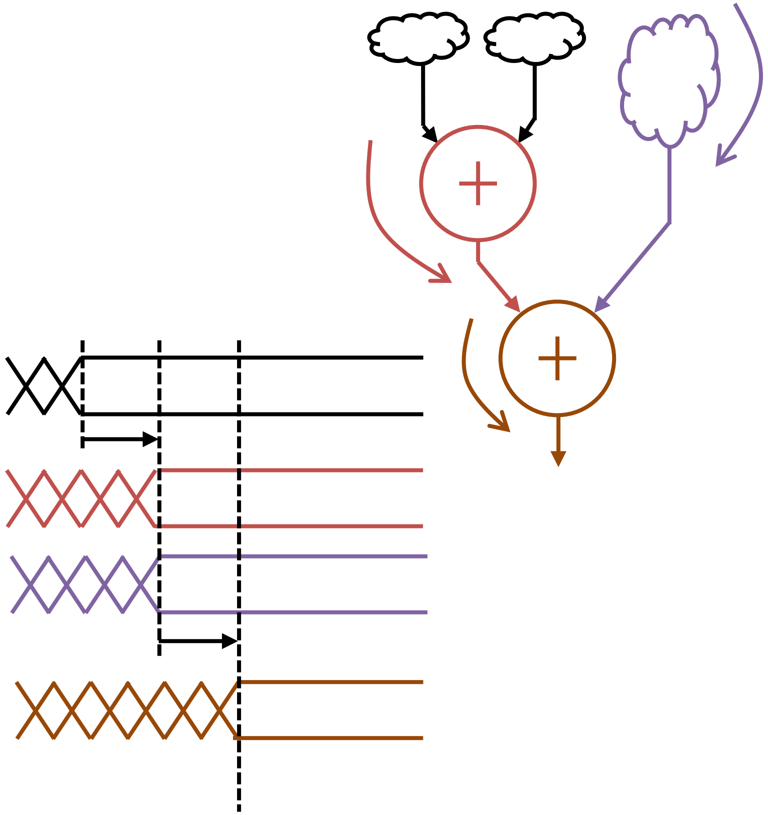
$(a + b) + (c + d)$



Algebraic Reordering for Timing

- **The longest path to the output of a datapath component (with identical delays from all its inputs) is determined by the latest arriving operand**
- **Operand arrival time is determined by**
  - **the delay of the previous logic and**
  - **the arrival time of the input to this previous logic**





- **Operand re-ordering assigns available slack to late arriving inputs**
  - **Late arriving inputs are used later in the datapath**

Area Reduction with Distributivity/Factoring

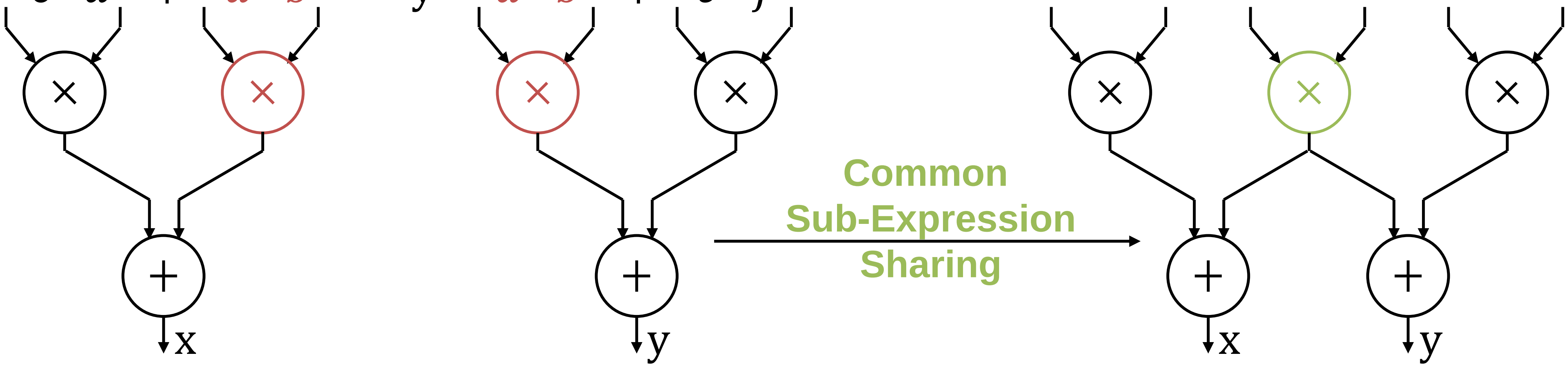
- **Objective:** reduce area by reducing the number of complex operations
- **Re-structuring / Factoring expressions:** re-order operations of different complexity to "share" complex operations
  - **Changing the order using distributivity**

$$a \cdot c \quad + \quad b \cdot c \qquad\qquad (a + b) \cdot c$$

Common Subexpression Sharing

- **Multiple results (outputs) often share common sub-expressions**
- **Common sub-expression sharing:** implement identical expressions (hardware) only once and re-use the result

$$x = c \cdot a + d \cdot b \qquad y = d \cdot b + e \cdot f$$



- **CAVE**: even if obvious, it is not uncommon to find the same operation multiple times (redundant) in a complex (especially hierarchical) HDL description
  - Synthesis tools can identify such common sub-expressions only if extremely obvious
  - Even irrelevant differences prevent automatic sharing of common sub-expressions

Identifying Common Subexpressions by Expansion

- **Common sub-expressions are not always immediately visible**
  - **Hidden especially after factoring expressions**
- **Expanding expressions helps to isolate common sub-expressions**
  - Expanding = Inverse operation of factoring

- **Example:** consider computation of three different results

$$x = (a + b) \cdot f \qquad y = (a + c) \cdot f \qquad z = (b + c) \cdot f$$

$$x = a \cdot f + b \cdot f \qquad y = a \cdot f + c \cdot f \qquad z = b \cdot f + c \cdot f$$

Three common sub-expressions: $a \cdot f$, $b \cdot f$, $c \cdot f$

  - This example: no immediately visible advantages

Identifying Trivial Common Subexpressions by Expansion

- **Common sub-expressions are not always immediately visible**
- **Expanding expressions helps to isolate common sub-expressions, BUT advantages are not always immediately visible**
  - **Number of operations is often not reduced or even increases**
- **HOWEVER: some sub-expressions (factors) can sometimes be significantly less complex than others, especially when constants are involved**
- **Example:** consider computation of three different results

Three full multiplications	$x = (a + 1) \cdot f$	$y = (a + 2) \cdot f$	$z = (a - 2) \cdot f$
Only ONE full multiplication	$x = a \cdot f + f$	$y = a \cdot f + 2 \cdot f$	$z = a \cdot f - 2 \cdot f$

Three common sub-expressions: $a \cdot f$, f, $2 \cdot f$

whereof two are trivial (constant multiplications with 1 and 2)

Special Operations and Special Cases

- **Some arithmetic operations can be realized without any hardware:**
- **Popular examples:**

  - **Multiplication by powers of two:** $x \cdot 2^N$ **left shift by *N* bits**
  - **Division by powers of two:** $x/2^N$ **right shift by *N* bits**
  - **Modulo with powers of two:** $x \,\%\, 2^N$ **keep only *N* least-significant bits**

- **Example:** serializing a 2D array index ($R \times C$) into a linear (1D) array of $R \cdot C$ with R rows and $C = 2^c$ columns

```
LinIDXxD <= RowIDXxD * #Columns + ColIDXxD
```

Note: only if number of `columns is power of 2`

```
signal RowIDXxD : unsigned(r-1 downto 0);
signal ColIDXxD : unsigned(c-1 downto 0);
signal LinIDXxD : unsigned(r+c-1 downto 0);
...
LinIDXxD <= RowIDXxD & ColIDXxD
```

Constant Operator Elimination & Operator Simplification

- **Operations between constants** are evaluated during synthesis and **require no dedicated hardware resources**
  - Operations between signals and constants can **not** be evaluated during synthesis

- **Constant isolation:** Re-ordering operations to group operations between only constants to avoid dedicated hardware

```
ZxDO <= AxDI + BxDI when (XxDI-1 > 5) else
        CxDI + DxDI;
```

```
ZxDO <= AxDI + BxDI when (XxDI > 5-1) else
        CxDI + DxDI;
```

- **Operator simplification:** modify constants to allow for using simpler operators
  - Modifying conditions for less complex hardware (timing & area)

```
CNTxDN <= CNTxDP + 1 when CNTxDP < 5 else
          (others => '0');
```

Note: only equivalent if `CNTxDP is never > 5`

```
CNTxDN <= (others => '0') when CNTxDP = 5 else
          CNTxDP + 1;
```

  - CAVEAT: sometimes, additional knowledge is required as modified conditions are not always fully equivalent without additional information (e.g., “counter will never exceed a specific value”)

Simplifying Expressions for Relational Operators

- **Conditions are often based on algebraic relational operators** $(<,>,\leq,\geq,=)$
- **Relational operators are invariant to** the application of the same **monotonous functions** to both of its parameters (sides)

$$\boldsymbol{x} \blacksquare \boldsymbol{y} \longleftrightarrow f(\boldsymbol{x}) \blacksquare f(\boldsymbol{y})$$

$\blacksquare$	$\{< \mid > \mid \leq \mid \geq \mid =\}$
$f(\cdot)$	monotonous function
$\boldsymbol{x}, \boldsymbol{y}$	algebraic expressions

- **Objective**: simplify the algebraic expressions $\boldsymbol{x}$ and $\boldsymbol{y}$ to reduce their complexity by finding an appropriate function $f(\cdot)$
- **Example:** checking if a point $[a, b]$ lies in a circle of a constant radius R

Note: highly complex square root operation

$$\sqrt{a^2 + b^2} < R \xrightarrow{f(x) = x^2} a^2 + b^2 < R^2$$

Note: no square root requires and R^2 is a constant

Approximations

- **Complex functions are often difficult to calculate precisely**
- **Hardware-friendly approximations often provide sufficient accuracy**

- **Example:** Euclidean norm $f(|a|,|b|) = \sqrt{a^2 + b^2}$

	$f(\lvert a\rvert,\lvert b\rvert)$
ℓ^1-norm	$\lvert a\rvert + \lvert b\rvert$
ℓ^∞-norm	$\max(\lvert a\rvert,\lvert b\rvert)$
Approx. 1	$\frac{3}{8}(\lvert a\rvert + \lvert b\rvert) + \frac{5}{8}\max(\lvert a\rvert,\lvert b\rvert)$
Approx. 2	$\max\left(\max(\lvert a\rvert,\lvert b\rvert), \frac{7}{8}\max(\lvert a\rvert,\lvert b\rvert) + \frac{1}{2}\min(\lvert a\rvert,\lvert b\rvert)\right)$